SYSTEMS AND METHODS FOR DYNAMICALLY LINKING APPLICATION SOFTWARE INTO A RUNNING OPERATING SYSTEM KERNEL

COPYRIGHT NOTICE

[001] A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights whatsoever.

COMPUTER PROGRAM LISTING APPENDIX

[002] A computer program listing appendix incorporating features of the present invention is being submitted herewith on a compact disc in compliance with 37 C.F.R. §1.52(e), and is incorporated herein by reference in its entirety. The computer program listing appendix is being submitted on a first compact disc labeled "Copy 1" and on a second compact disc labeled "Copy 2." The disc labeled Copy 2 is an exact duplicate of the disc labeled Copy 1. The files contained on each disc are: (1) user.c (size = 9406 bytes, creation date = 07/10/2003); (2) Makefile (size = 737 bytes, creation date = 07/10/2003); (3) rtl_crt0.c (size = 7151 bytes, creation date = 07/10/2003); (4) example.c (size = 972 bytes, creation date = 09/24/2003); (5)

rtl_mainhelper.h (size = 3847 bytes, creation date = 07/10/2003); and (6) rtl_mainhelper.c (size = 5438 bytes, creation date = 07/10/2003). The computer program listing and the files contained on the compact discs are subject to copyright protection and any use thereof, other than as part of the reproduction of the patent document or the patent disclosure, is strictly prohibited.

BACKGROUND OF THE INVENTION

1. Field of the Invention

[003] The present invention relates, generally, to systems and methods for dynamically linking application software into a running operating system kernel and, more specifically, to systems, software, hardware, products, and processes for use by programmers in creating such application software, and dynamically loading it into and unloading it from a running operating system kernel.

2. Discussion of the Background

[004] There are many software applications in which it is desirable to link the software or one or more modules of the software into a running operating system kernel. For example, it is generally desirable to link device driver software into the operating system kernel because it enables the device driver software to access kernel data space.

[005] Many operating systems offer utilities for dynamically linking application software (a.k.a, "application code" or "application modules") into a running operating system kernel. Typically, these methods involve copying the application module code and data into kernel memory and then resolving symbol table references. Conventionally, application modules that are to be linked into an operating system kernel include the following sections of code: (1) a section of operating system "kernel" code, (2) a section of code to handle initialization of the module, and (3) a section of code to handle module cleanup when the module is in the process of being removed.

[006] These application modules are considered, essentially, to be dynamically loadable parts of the operating system kernel. But operating system kernels provide a very complex and low level interface to these application software modules, complicating the process of developing such application software and requiring the programmer to directly address and write operating system kernel code, code to handle initialization/loading of the application software, and code to handle unloading/cleanup after termination. Furthermore, there is generally no defined and stable application programming interface within an operating system, and required data structures and interfaces change especially rapidly in operating systems, such as, for example, BSD and Linux, further complicating the programmer's

development and implementation of the application software. Creating and loading/unloading the modules is often complex, time consuming, and requires constant monitoring and updating of the code to ensure reliable, error-free implementation.

[007] What is desired, therefore, are systems and methods to overcome the above described and other disadvantages of the conventional system and methods for dynamically linking modules into a running operating system kernel.

SUMMARY OF THE INVENTION

[008] The present invention provides systems and methods for dynamically linking modules into a running operating system kernel. The systems and methods of the present invention overcome the above described and other disadvantages of the conventional systems and methods. For example, the systems and methods specified herein (1) permit an application programmer to write, compile, execute, and terminate application code that is to be loaded into a kernel as if the application code was an ordinary (i.e., not kernel loadable) application program, (2) allow a standard programming environment to be used to encapsulate application software in a familiar environment, and (3) permit automatic cleanup of errors and freeing of program resources when the application terminates. The present invention preserves the advantages of in-

kernel programming by providing an application program access to the hardware address space seen by the kernel, as well as access to kernel data structures, internal kernel services and to privileged machine instructions.

[009] Advantageously, the present invention can be applied to a wide range of application environments (what the application programmer sees) and kernel environments (the underlying programming environment). For example, we have implemented the method for the POSIX threads application environment and the RTLinux and RTCore BSD kernel environments. The RTLinux kernel environment is described in U.S. Patent No. 5,995,745, the contents of which are incorporated herein by reference.

[0010] In one embodiment, the system of the present invention enables a programmer to dynamically link application code created by the programmer into a running operating system kernel , wherein the system includes the following components: (1) an environment library comprising one or more routines for insulating the application code from the operating system environment and for implementing a uniform execution environment; (2) a build system for constructing a loadable module from the application code and the environment library; (3) an execution library comprising one or more routines for encapsulating the loadable module within a standard executable program file, transparently loading the loadable module into

the running operating system kernel, setting up input/output channels that may be required, passing arguments to the loadable module, and terminating and unloading the loadable module after receiving a termination signal; (4) an infrastructure library comprising one or more routines that may need to be executed prior to loading the loadable module into the running operating system kernel and/or after unloading the loadable module from the kernel (such routines may include routines to allocate stable memory (memory that will need to be held after the module completes and is unloaded), routines to initialize a list of modules that will be managed, routines to free memory used by the module, and routines to close files and free semaphores and other resources used by the module); and (5) a build system for constructing the executable program from the loadable module and the execution library, wherein the executable program may be in several files or a single file.

[0011] In another aspect, the present invention includes a computer readable medium, such as, for example, an optical or magnetic data storage device, having stored thereon the execution library, the environment library, the infrastructure library, and the build system.

[0012] The above and other features and advantages of the present invention, as well as the structure and operation of preferred embodiments of the present

invention, are described in detail below with reference to the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

[0013] The accompanying drawings, which are incorporated herein and form part of the specification, illustrate various embodiments of the present invention and, together with the description, further serve to explain the principles of the invention and to enable a person skilled in the pertinent art to make and use the invention. In the drawings, like reference numbers indicate identical or functionally similar elements. Additionally, the left-most digit(s) of a reference number identifies the drawing in which the reference number first appears.

[0014] FIG. 1 is a functional block diagram that illustrates the components of a system according to one embodiment of the invention.

[0015] FIG. 2 is a diagram illustrating a first function of a build system component of the invention.

[0016] FIG. 3 is a diagram illustrating a second function of a build system component of the invention.

[0017] FIG. 4 is a flow chart illustrating a process 400 that may be performed by build system component of the invention.

[0018] FIG. 5 is a flow chart illustrating a process performed by the "main" routine of the execution library component of the invention.

[0019] FIG. 6 illustrates example pseudo-code of an example loadable module.

[0020] FIG. 7 illustrates a representative computer system for implementing the systems and methods of the present invention for dynamically linking application software into a running operating system kernel.

DETAILED DESCRIPTION OF THE PREFERRED EMBODMENT

[0021] In the following description, for purposes of explanation and not limitation, specific details are set forth, such as particular systems, computers, devices, components, techniques, computer languages, storage techniques, software products and systems, operating systems, interfaces, hardware, etc. in order to provide a thorough understanding of the present invention. However, it will be apparent to one skilled in the art that the present invention may be practiced in other embodiments that depart from these specific details. Detailed descriptions of well-known systems, computers, devices, components, techniques, computer languages, storage techniques, software products and systems, operating systems, interfaces, and hardware are omitted so as not to obscure the description of the present invention.

[0022] FIG. 1 is a functional block diagram that illustrates the components of a system 100, according to one embodiment of the invention, for dynamically linking application code 102 into a running operating system kernel 104. Application code 102 is code written by a user 101 of system 100. In one embodiment, the code is written as a standard "C" program and includes the "main" function. Example application code that can be used with system 100 is provided in the file "example.c", which is included in the above referenced computer program listing appendix.

[0023] As shown in FIG. 1, system 100 includes an infrastructure library 110, an environment library 112, an execution library 114, and a build system 116. As used herein, the term "library" means: "a set of one or more routines," and the term "routine" means: "a set of one or more program instructions that can be executed." Preferably, each library 112, 114, and 116 is stored in a separate file, but this need not be the case. The libraries could all be stored in one file, in multiple files, or in any other suitable storage architecture or arrangement.

[0024] Build system 116 is configured to construct a "loadable module" 202 from application code 102 and environment library 112, as is illustrated in FIG. 2. As used herein the term "loadable module" means "object code produced from the application code and environment library." In one embodiment, the build

system 116 includes one or more makefiles. As further illustrated in FIG. 3, build system 116 is configured to construct an executable program 302 from loadable module 202 and execution library 114.

[0025] FIG. 4 is a flow chart illustrating a process 400 that may be performed by build system 116 to create loadable module 202 and executable program 302. Process 400 can be used when, for example, the C programming language is used in implementing the invention or with any other comparable programming language. Process 400 begins in step 402, where build system 116 compiles application code 102 into object code. Next (step 404), build system 116 links the object code with the environment library 112 object code to produce a linked object module. Next (step 406), build system 116 converts the linked object module into a C code array. This C code array is the "loadable module." Next (step 408), build system 116 compiles the C code array produced in step 406 to produce an object file. Next (step 410), system 116 links the object file produced in step 408 with execution library 112 object code to produce the executable program 302.

[0026] In one embodiment, when user 101 executes program 302, a routine from the execution library 114 sets up an input channel and an output channel by connecting the standard output and standard input of the original process to the standard output and input of the process that will insert the module, and to the

fifos that connect user space to kernel space. Setting up the input/output channels is optional if application code 102 does not use input/output channels. Also, when program 302 is executed a routine from execution library 114 inserts loadable module 202 into the operating system address space. Once loadable module 202 is inserted into the operating system address space, loadable module 202 begins to execute.

[0027] If application code 102, which is included in loadable module 202, uses input/output channels, then a routine from execution library 114 waits for loadable module 202 to connect via kernel/user channels, which are implemented as RT-fifos in this embodiment, and then connects those kernel/user channels to the input/output channels mentioned above.

[0028] After loadable module 202 begins to execute, a routine from environment library 112 creates kernel/user channels, creates a thread to execute application code 102, and then waits for the thread to complete. Creating the kernel/user channels is optional. When the thread completes, a routine from environment library 112 frees resources and unloads loadable module 202. The routines from environment library 112 may need to call routines from infrastructure library 110 for carrying out their work. For example, environment library 112 may delegate some cleanup operations to infrastructure

library 110 so that cleanup can take place after loadable module 202 has been unloaded.

EXAMPLE CODE FOR IMPLEMENTING THE INVENTION

[0029] The following examples of code are for the Linux, RTLinux, and/or RTCore BSD operating systems. However, the present invention is not limited to operating only within the Linux operating system. The present invention can, for example, work with other operating systems, including: Net BSD and Free BSD, Apple OS X, other UNIX type operating systems, WindRiver's VxWorks system, and Microsoft's XP and NT operating systems.

[0030] Example code for implementing execution library 114 is provided in the file "user.c", which is included in the above referenced computer program listing appendix. This code is merely an example and should not be used to limit the invention.

[0031] The example code for implementing execution library 114 is written in the C language and includes a "main" routine. This main routine is the first routine that is executed after user 101 executes executable program 302. FIG. 5 is a flow chart illustrating the process 500 performed by the main routine. As shown in FIG. 5, the main routine collects the arguments to be passed to loadable module 202 (step 502). Next (step 504), the main routine sets up the input and output channels. Next (step 506), main routine creates a child process by

executing the fork() routine. The child process immediately replaces its process image with the insmod process image. The child process does this by executing the execl() routine.

[0032] Next (step 508), the main routine pipes loadable module 202 to the insmod process. This causes the insmod process to put loadable module 202 with arguments in kernel address space. When the application code within loadable module 202 begins running in kernel space, a routine within loadable module 202 will create kernel/user channels (e.g., RTCore "fifos") that move data between kernel and user space.

[0033] The main routine waits until it can open these kernel/user channels (step 510). Once it can open the channels it does and then uses the channels to transfer data back and forth so that data coming from loadable module 202 is sent to the standard output of executable program 302 and data coming into the standard input of executable program 302 is transmitted via the kernel/user channel to loadable module 202 (step 512).

[0034] Example code for implementing environment library 112 is provided in the file "rtl_crt0.c", which is included in the above referenced computer program listing appendix. This code is merely an example and should not be used to limit the invention.

[0035] As discussed above, environment library 112 includes one or more routines for insulating application code 102 from the operating system environment and implementing a uniform execution environment. That is, instead of depending on a changeable and specialized operating system interface, the program can use standard "C" interfaces - or alternatively, any standard application programming interface. For example, in the current embodiment of the invention, instead of using a Linux kernel "sys_open" operation applied to some terminal device so that the program can output data, the program can simply use the standard "printf" routine. Instead of using some OS dependent routine for generating a thread, the program can use "pthread_create" - a POSIX standard function.

[0036] As shown in the example code, environment library 112 includes an initialization routine called "init_module()" for performing the functions of insulating application code 102 from the operating system environment and implementing a uniform execution environment. The init_module routine is executed as soon as the module is loaded into the kernel address space and is executed in the context of the process that invokes the insert/load operation. The init_module performs the following steps: (1) copies in arguments that have been passed to it by the execution library and that were passed to the execution library by the user or program that invoked

the process, (2) creates the kernel/user channels that connect loadable module 202 to the executable program 302, (3) requests a block of memory from the operating system and stores a "task" structure that describes the application in the module to the infrastructure library, (4) puts the data describing application code 102 on a "task" list that is used by the infrastructure library to control all the modules that it manages, and (5) creates a Linux kernel thread to run a "startup_thread" routine, which is included in the infrastructure library 110 and which is describe below.

[0037] As further shown, environment library 112 may also include a cleanup routine. The cleanup routine included in the example environment library is called "cleanup_module." The cleanup_module routine performs the following task: (1) removes from the task list the task (i.e., the data describing application code 102; (2) waits for the kernel thread to terminate; (3) closes the channels created by the init_module; (4) cleans up state by freeing memory, closing files, and releasing semaphores and possibly other resources; and (5) frees the block of memory that was used to store the task structure.

[0038] Example code for implementing infrastructure library 110 is provided in the files "mainhelper.c" and "mainhelper.h", which are included in the above referenced computer program listing appendix. This

code is merely an example and should not be used to limit the invention.

[0039] The example infrastructure library 110 includes the "startup_thread" routine. As discussed above, the init_module routine creates a Linux kernel thread to run the startup_thread routine. As is illustrated from the example code, the startup_thread routine does some operating system specific setup by detaching the kernel process from any terminals ("daemonize" in Linux), and filling in information in the task structure such as the identity of the current kernel thread, and then executes application code 102 by calling the "task" function. After executing application code 102, the startup_thread routine waits for application code 102 to terminate. When application code 102 terminates, the startup_thread routine sends the application code 102 return value down the output channel, signals completion, and exits. By signaling completion, the startup_thread routine causes the cleanup_module routine to execute.

[0040] The example infrastructure module 110 further includes a routine called "rtl_main_wait." This routine implements the function to wait for the application kernel thread to complete. The function "rtl_main_wait" is called in the application after threads are started so that the "main" routine can safely be suspended until the subsidiary threads are completed.

[0041] Example code for implementing build system 116 is provided in the file "Makefile", which is included in the above referenced computer program listing appendix. This code is merely an example and should not be used to limit the invention.

[0042] EXAMPLE APPLICATIONS OF THE INVENTION

[0043] In general, all of the modules described below require the addition of application code to a running or booting operating system - the basic functionality provided by loadable kernel modules. Utilizing the present invention, the modules are all (1) simpler to develop and debug, (2) simpler to connect to other components to make a useful software system, and (3) more reliable. Since development time and software reusability (connection to existing software applications) dominate the costs of producing software, the invention provides a significant economic value. In addition, the present invention provides the programmer with a familiar, less complicated development environment, and does not require the programmer to handle complicated kernel-related initialization/loading and unloading/cleanup functions. The invention, thus, facilitates more reliable, error-free application code.

[0044] (1) A real-time data acquisition module under RTLinux:

[0045] The present invention enables a loadable kernel data acquisition module to be developed and

tested as if the module were a standard non-kernel module. To develop a real-time data acquisition system in RTLinux using the invention all one needs to do is write a program that runs under the UNIX operating system. The program should include a main program and a thread. The thread should be configured to sample data from the device from which data is to be acquired (e.g., a voltage sensor) at a fixed interval and then write the data to standard output. The program text in pseudo-code 600 is shown in FIG. 6.

[0046] This program can be tested for logical correctness under any POSIX compliant standard UNIX. After testing for logical correctness, the program can be rebuilt using build system 116 to create an executable program that can be run under RTLinux or RTCore BSD, for example. The complexity of launching the thread into the real-time operating system, connecting the thread to data streams, and closing the module on termination is now all hidden to the application. The module can be tested on a UNIX command line with the command: % data_acquisition > test_file.

[0047] (2) A device driver module:

[0048] As noted herein, the invention provides a means of simplifying and automating module deployment and testing. Currently, the most common use for loadable kernel modules is for device drivers that are not statically linked into the operating system.

Drivers in loadable kernel module form allow an operating system to configure itself on boot - adding drivers for devices it detects - and allowing users and system administrators to update systems by, for example, upgrading a disk drive or a communications device and adding a new driver without re-booting a system and interrupting other functions.

[0049] However, driver modules are often very dependent on specific releases of the operating system and are notorious for failing to cooperate with other components. Using the invention, a driver developer can insulate the driver from non-relevant operating system internal changes. More importantly, a system administrator can rely on the automatic cleanup provided by the invention to improve reliability and rely on the automatic setup of communication channels to improve reports. In a Linux system, without the invention, the method for testing a loadable kernel driver module might involve the following steps: (1) the administrator logs in as root to get the correct privileges; (2) the administrator uses the "insmod" utility to attempt to load the driver; and (3) if the insmod utility works, the system administrator uses a "dmesg" utility to print out internal operating system diagnostic messages and to see if he or she can find one relevant to the driver. Suppose that the driver cannot correctly start because, for example, a prior driver for the device has not been removed. At this point, the system administrator must use the "rmmod"

tool to try to remove both the new driver and the prior one, and then again tries to install the new driver. The driver writer must have correctly handled the complex case of responding to an "rmmod" request.

[0050] With the invention, the method for testing the loadable kernel driver is much simpler. The method might include simply the step of running the executable program that loads the module into the kernel. If the module needs to be unloaded from the kernel, the administrator need only execute the kill command to kill the executable program.

[0051] (3) An encryption module:

[0052] This example illustrates the value of the automatic creation of input/output channels by the invention. Suppose that we have a generic operating system module that provides an encryption and security stamp facility and want to attach it to a database. The command line for initiating secure operation of the database in a system utilizing the invention might be:

```
%(decrypt_input | my_database | encrypt_output)&
```

so that the two security modules (i.e., "decrypt_input" and "encrypt_output") are automatically loaded and connected to the inputs and outputs of database module, "my_database". The entire system can be terminated and automatically unloaded with a single signal.

[0053] (4) A security module:

[0054] Loadable kernel modules are themselves potentially a security weakness of an operating system, since modules traditionally operate within the address space and with all privileges of the operating system itself. There is generally only a check on whether the user loading the module has sufficient privileges to load a module. However, the invention makes it convenient to add more sophisticated security checking either directly in the infrastructure libraries or in a root module that controls all module loads after loading. This module can validate certificates and even provide dynamic code check.

[0055] (5) A fault tolerant module:

[0056] The invention provides a means of dynamically adding a fault tolerance capability to a running operating system by adding a kernel data logger. For example, the script % checkpoint_kernel | netcat 10.0.0.244:45 runs a fault tolerance module named "checkpoint_kernel" and sends the module's output to a standard program (i.e., "netcat") that directs output to a named internet site and TCP port. Using prior methods, the programmer would have had to hand code creation of an output channel in the checkpoint_kernel module and then produce further code to redirect the data to a destination and to process the destination IP and port as parameters.

[0057] FIG. 7 is an illustration of a representative computer system for implementing the systems and methods of the present invention for

dynamically linking application software into a running operating system kernel. With reference to FIG. 7, the method of the present invention may be advantageously implemented using one or more computer programs executing on a computer system 702 having a processor or central processing unit 704, such as, for example, a workstation, server, or embedded-single-board computer using, for example, an Intel-based CPU, such a Centrino, running one of the operating systems previously described, having a memory 706, such as, for example, a hard drive, RAM, ROM, a compact disc, magneto-optical storage device, and/or fixed or removable media, having a one or more user interface devices 708, such as, for example, computer terminals, personal computers, laptop computers, and/or handheld devices, with an input means, such as, for example, a keyboard 710, mouse, pointing device, and/or microphone. The computer program is stored in memory 11 along with other parameters and data necessary to implement the method of the present invention.

[0058] In addition, the computer system 702 may include an analog-to-digital converter, sensors, and various input-output devices, and may be coupled to a computer network, which may also be communicatively coupled to the Internet and/or other computer network to facilitate data transfer and operator control.

[0059] The systems, processes, and components set forth in the present description may be implemented using one or more general purpose computers,

microprocessors, or the like programmed according to the teachings of the present specification, as will be appreciated by those skilled in the relevant art(s). Appropriate software coding can readily be prepared by skilled programmers based on the teachings of the present disclosure, as will be apparent to those skilled in the relevant art(s). The present invention thus also includes a computer-based product which may be hosted on a storage medium and include instructions that can be used to program a computer to perform a process in accordance with the present invention. The storage medium can include, but is not limited to, any type of disk including a floppy disk, optical disk, CDROM, magneto-optical disk, ROMs, RAMs, EPROMs, EEPROMs, flash memory, magnetic or optical cards, or any type of media suitable for storing electronic instructions, either locally or remotely.

[0060] While the processes described herein have been illustrated as a series or sequence of steps, the steps need not necessarily be performed in the order described, unless indicated otherwise.

[0061] The foregoing has described the principles, embodiments, and modes of operation of the present invention. However, the invention should not be construed as being limited to the particular embodiments described above, as they should be regarded as being illustrative and not as restrictive. It should be appreciated that variations may be made in those embodiments by those skilled in the art

without departing from the scope of the present invention. Obviously, numerous modifications and variations of the present invention are possible in light of the above teachings. It is therefore to be understood that the invention may be practiced otherwise than as specifically described herein.

[0062] Thus, the breadth and scope of the present invention should not be limited by any of the above-described exemplary embodiments, but should be defined only in accordance with the following claims and their equivalents.